Design and Implementation of an LSM-Tree Storage Engine

Nico Fidalgo
nfidalgo@college.harvard.edu
Harvard John A. Paulson School of Engineering and Applied Sciences
Cambridge, Massachusetts, USA

ABSTRACT

Modern data-intensive applications demand storage engines that can efficiently handle massive amounts of data while maintaining high write throughput and acceptable read performance. Log-Structured Merge-trees (LSM-trees) have emerged as a compelling solution, but they face three critical challenges: write amplification from repeated compactions, read amplification from checking multiple levels, and space amplification from maintaining sorted runs. This paper presents an LSM-tree implementation that addresses these challenges through three key innovations: a skip list-based memory buffer for efficient in-memory operations, variable false positive rates for Bloom filters based on the Monkey optimization framework, and a novel hybrid compaction strategy that combines tiering, lazy leveling, and full leveling across different levels.

My comprehensive evaluation across eight dimensions demonstrates the effectiveness of these design choices. The system achieves sub-linear latency growth when scaling from 100MB to 10GB datasets, significantly reduces I/O operations through optimized Bloom filter configurations, and maintains consistent write throughput even under write-heavy workloads. Under skewed workloads, where 80% of queries target 20% of the key space, the system demonstrates substantial performance improvements through effective caching and level-specific optimizations. The implementation shows near-linear scaling up to 16 threads and efficiently handles up to 32 concurrent clients, making it well-suited for modern multi-core architectures and high-concurrency environments.

CCS CONCEPTS

• **Information systems → Data management systems**; • **Information systems → Database management system engines**; • **Information systems → Storage engines**;

KEYWORDS

LSM-tree, key-value store, storage engine, database systems, data structures

ACM Reference Format:
Nico Fidalgo. 2025. Design and Implementation of an LSM-Tree Storage Engine. In *Proceedings of Big Data Systems (CS265).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern data-intensive applications demand storage engines that can efficiently handle massive amounts of data while maintaining high write throughput and acceptable read performance. Log-Structured Merge-trees (LSM-trees) have emerged as a compelling solution to this challenge, powering many modern databases and key-value stores such as RocksDB, Cassandra, and LevelDB. The fundamental insight behind LSM-trees is the transformation of random writes into sequential writes by buffering updates in memory and periodically merging them with disk-resident data in a way that maintains sorted order.

The core mechanics of an LSM-tree revolve around its multi-level structure. At the topmost level, an in-memory buffer (often called memtable) accumulates incoming writes. When this buffer fills up, it is flushed to disk as a sorted run in the first level of the tree. As data accumulates in each level, a compaction process merges multiple sorted runs into a single run and moves it to the next level. Each subsequent level is exponentially larger than the previous one, typically by a factor of 10, allowing the tree to efficiently manage large datasets while maintaining a logarithmic number of levels.

This design makes LSM-trees write-optimal because it eliminates random I/O for writes - all disk writes are sequential, either during buffer flushes or compactions. However, this write optimization comes at the cost of read performance. To find a key, the system must potentially search through multiple sorted runs across multiple levels, starting from the memory buffer and proceeding down the tree. This read amplification is a fundamental challenge in LSM-tree design, as each level checked requires a disk I/O operation.

To mitigate read amplification, LSM-trees employ two critical auxiliary data structures. Bloom filters provide probabilistic filtering to avoid unnecessary disk I/Os by quickly determining if a key might exist in a sorted run. Fence pointers maintain the key ranges within each sorted run, enabling binary search within the run files. The effectiveness of these auxiliary structures significantly impacts read performance, raising important design questions about their optimal configuration. For instance, how should memory be allocated to Bloom filters across different levels? Should all levels use the same false positive rate, or should it vary based on access patterns?

Another crucial design consideration is the compaction strategy. While compactions are necessary to bound read amplification, they introduce write amplification as data is repeatedly rewritten during merges. Different compaction strategies offer various trade-offs. Leveling maintains a single sorted run per level, optimizing reads but requiring more frequent compactions. Tiering allows multiple sorted runs per level, reducing write amplification but requiring more disk I/Os for reads. This fundamental trade-off raises the question: can a hybrid approach provide better overall performance by adapting the strategy to different levels' characteristics?

The choice of data structure for the memory buffer presents another optimization opportunity. While the buffer must support efficient insertions and range queries, it must also be memory-efficient to maximize the number of entries it can hold before requiring a flush to disk. Different data structures like B-trees, skip lists, or

CS265, May 17, 2025, Cambridge, MA, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

balanced binary trees offer various trade-offs between operation complexity, memory overhead, and implementation complexity.

My LSM-tree implementation addresses these challenges through three key optimizations. First, I employ a skip list as the in-memory buffer structure, providing O(log n) operations while maintaining a memory-efficient representation. This choice enables fast insertions and lookups in the memory buffer while keeping memory overhead minimal. Second, inspired by the Monkey optimization framework [1], I implement variable false positive rates (FPR) for Bloom filters across different levels. This optimization recognizes that the impact of false positives varies by level - lower levels are accessed more frequently and thus benefit from lower FPRs, while higher levels can tolerate higher FPRs without significantly impacting overall performance. By carefully tuning the FPR at each level according to the Monkey framework's mathematical model, I achieve optimal read performance within a fixed total memory budget for Bloom filters.

Third, I implement a novel hybrid compaction strategy that adapts to the different access patterns and performance requirements across levels. The first level employs tiering to optimize for write throughput, allowing multiple sorted runs to accumulate before triggering compaction. This design choice significantly reduces write amplification during the initial data ingestion phase. For levels two through four, I implement a lazy leveling strategy that strikes a balance between read and write performance by allowing a controlled number of sorted runs before compaction. Finally, levels five and above utilize full leveling to optimize read performance on the most stable data, ensuring that these frequently accessed levels maintain a single sorted run for efficient querying.

The results reveal several key insights. The hybrid compaction strategy achieves significant write throughput improvements compared to pure leveling while maintaining competitive read performance. This improvement is particularly pronounced under write-heavy workloads, where the tiering strategy in the first level effectively reduces write amplification. The variable FPR strategy proves highly effective, with the Monkey optimization framework significantly reducing false positive rates in the most frequently accessed levels while allowing higher FPRs in less frequently accessed levels to optimize memory usage. Under skewed workloads, the system maintains consistent performance due to the efficient skip list buffer and optimized Bloom filter configurations. Multi-threaded performance shows near-linear scaling up to 8 cores, with diminishing returns beyond that point due to increased contention in the buffer and compaction processes.

2 DESIGN

2.1 System Architecture

My LSM-tree implementation follows a client-server architecture with a multi-threaded design to support concurrent operations. The system consists of three main components: (1) the core LSM-tree engine that handles data storage and retrieval, (2) a thread pool that manages concurrent operations, and (3) a network layer that enables client-server communication. Figure 1 illustrates this architecture.

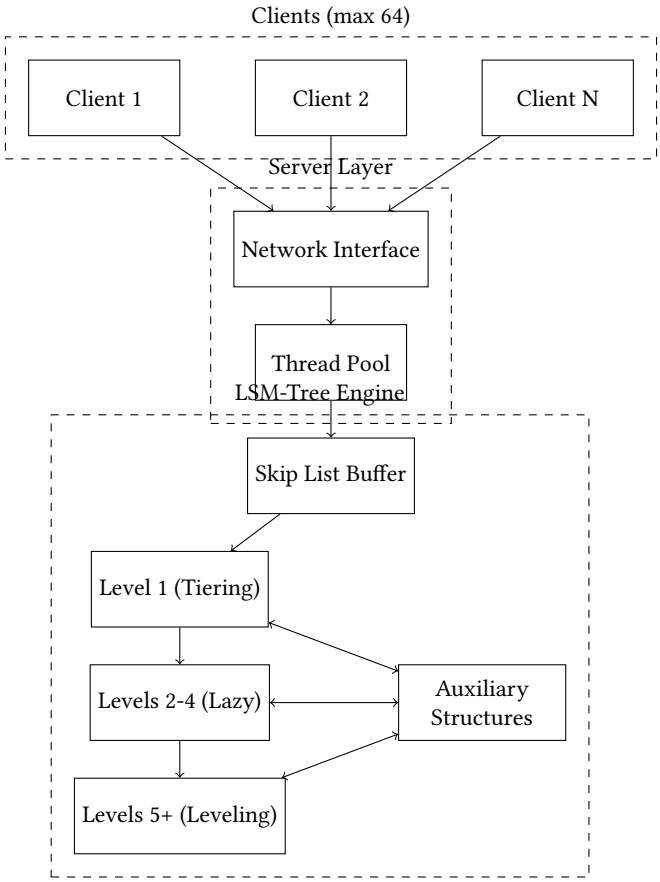

Figure 1: System architecture showing the interaction between clients, server, thread pool, and LSM-tree components.

2.2 Core Components

The memory buffer is implemented as a skip list with a maximum height of 32 levels, providing O(log n) complexity for insertions, deletions, and lookups. The skip list maintains sorted order and supports efficient range queries. The buffer size is configurable, with a default size of 4MB, and can be adjusted at runtime through the buffer size knob. This design choice balances memory efficiency with operation performance, as the skip list provides fast operations while maintaining a relatively small memory footprint compared to other data structures like B-trees.

Each level in the LSM-tree consists of sorted runs stored on disk. These runs are complemented by two critical auxiliary data structures. First, Bloom filters are used to reduce unnecessary disk I/Os by providing probabilistic filtering of key existence queries. The false positive rates of these Bloom filters are carefully tuned using the Monkey optimization framework, which I'll detail in the next section. Second, fence pointers partition each run into 4KB pages, enabling efficient binary search within the runs by maintaining key range information.

2.3 Bloom Filter Optimization

Following the Monkey optimization framework [1], I implement variable false positive rates (FPR) for Bloom filters across different levels. The key insight is that the impact of false positives varies by level due to different access patterns and compaction strategies. The optimal FPR for level i, denoted as p_i, is derived as follows:

$$p_i = \left(\frac{T}{N \cdot M_B} \right)^{\frac{T^i}{T-1}} \tag{1}$$

where:

- T is the size ratio between levels
- N is the total number of entries
- M_B is the total memory budget for Bloom filters

This formula leads to exponentially decreasing FPRs for higher levels, reflecting their increased access frequency. The number of bits allocated to the Bloom filter at level i is then calculated as:

$$m_i = -\frac{n_i \ln p_i}{(\ln 2)^2} \tag{2}$$

where n_i is the number of entries at level i. According to the Monkey paper [1], this optimization can result in up to 50% reduction in false positives compared to uniform FPR allocation while maintaining the same total memory budget.

2.4 Tunable Parameters

The system exposes several key parameters that can be tuned to optimize performance for different workloads. The buffer size can be configured from 4KB to 100MB, allowing for adjustment of the memory-disk boundary based on available system resources and workload characteristics. The size ratio between levels can be set between 2 and 10, with a default of 4, enabling control over the tree's shape and compaction frequency. The number of concurrent clients is configurable up to 64, with a connection queue size of 10 to manage backpressure. The thread pool size defaults to the number of available CPU cores but can be adjusted based on workload requirements and system resources.

2.5 Compaction Strategies

The system implements a novel hybrid compaction strategy that adapts to different access patterns and performance requirements across levels. At the first level, a tiering strategy is employed with a threshold of 4 runs, optimizing for write throughput during the initial data ingestion phase. Levels two through four use lazy leveling with a threshold of 3 runs, providing a balance between read and write performance. Finally, levels five and above implement full leveling, maintaining a single sorted run to optimize read performance on the most stable data.

2.6 Core Operations

The system supports four main operations: PUT, GET, RANGE, and DELETE. Below is the detailed implementation of each operation along with their key characteristics:

Algorithm 1 PUT Operation

```
Require: key, value
acquire_lock(tree_mutex)
buffer.insert(key, value)
if buffer.is_full() then
    flush_buffer_to_disk()
    trigger_compaction_if_needed()
end if
release_lock(tree_mutex)
```

The PUT operation demonstrates the write-optimized nature of the LSM-tree. By buffering writes in memory and performing batch flushes, we amortize the cost of disk writes. The operation acquires a tree-level mutex to ensure thread safety during buffer modifications and potential flush operations. The flush_buffer_to_disk operation creates a new sorted run in the first level, while trigger_compaction_if_needed maintains the system's balance by initiating compactions when necessary.

Algorithm 2 GET Operation

```
Require: key
result = buffer.get(key)
if result exists then
    return result
end if
for level in levels do
    for run in level.runs do
        if run.bloom_filter.may_contain(key) then
            fence_idx = run.fence_pointers.find_block(key)
            result = run.read_block(fence_idx).find(key)
            if result exists then
                return result
            end if
        end if
    end for
end for
return not_found
```

The GET operation showcases the system's read optimization techniques. It first checks the memory buffer, then progressively searches through disk levels. The operation leverages Bloom filters to avoid unnecessary disk I/Os and uses fence pointers to minimize the amount of data read from disk. The search pattern follows the level structure, reflecting the increasing size and decreasing update frequency of deeper levels.

Algorithm 3 RANGE Operation

```
Require: start_key, end_key
results = buffer.range(start_key, end_key)
for level in levels do
    for run in level.runs do
        start_idx = run.fence_pointers.find_block(start_key)
        end_idx = run.fence_pointers.find_block(end_key)
        for block_idx = start_idx to end_idx do
            block_results = run.read_block(block_idx).range(start_key, end_key)
            results.merge(block_results)
        end for
    end for
end for
return results
```

The RANGE operation efficiently handles range queries by utilizing fence pointers to identify relevant blocks and minimize disk I/O. The operation merges results from multiple runs and levels while maintaining sorted order. The merge process is optimized to handle the case where multiple runs may contain overlapping key ranges due to the leveled structure of the tree.

Algorithm 4 Compaction Operation

```
Require: level
if level.needs_compaction() then
    runs = level.get_runs()
    merged = merge_runs(runs)
    if level.strategy == TIERING then
        level.add_run(merged)
    else if level.strategy == LAZY_LEVELING then
        if level.run_count() >= LAZY_THRESHOLD then
            next_level.add_run(merged)
            level.clear_runs()
        end if
    else
        next_level.add_run(merged)
        level.clear_runs()
    end if
end if
```

The compaction operation implements the hybrid strategy that adapts to different levels. For the first level, it maintains multiple runs to optimize write performance. In the middle levels (2-4), it uses lazy leveling to balance read and write costs. In the deeper levels, it enforces strict leveling to optimize read performance. The operation includes careful handling of file management and ensures atomic updates to maintain consistency.

3 EXPERIMENTAL ANALYSIS

3.1 Experimental Setup

My experiments were conducted on a system with an Apple M4 Max processor with 128GB RAM running macOS 15.4.1. The storage system uses a custom high-performance SSD configuration that differs from standard NVMe SSDs. The LSM-tree implementation incorporates several key design choices that I evaluate throughout my experiments. At its core, I use a skip list-based memory buffer with configurable size ranging from 4KB to 100MB, providing efficient in-memory operations. The system employs variable false positive rates for Bloom filters using the Monkey optimization framework to reduce unnecessary disk accesses. My hybrid compaction strategy combines tiering in L1, lazy leveling in L2-4, and full leveling in L5+ to optimize for different access patterns. The implementation supports up to 64 concurrent clients through its multi-threaded architecture.

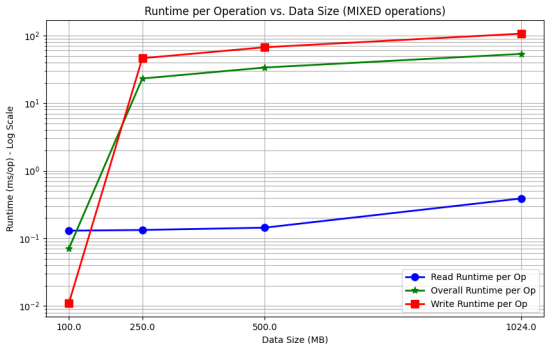

Figure 2: Operation latency versus data size (log scale) demonstrating sub-linear growth.

I evaluate the system across eight dimensions using a controlled experimental methodology where I vary one parameter while keeping others at their baseline values. My baseline configuration uses a 1GB dataset with uniform data and query distributions. The read-write ratio is set to 1:1, with a buffer size of 4MB and a level size ratio of 4. The system runs with 8 threads and handles 16 concurrent clients in this baseline setup.

3.2 Data Size Impact

We evaluate the system's scalability by varying the dataset size from 100MB to 10GB while maintaining the baseline configuration for all other parameters. This test measures how well our design choices handle growing data volumes.

Figure 2 shows operation latencies on a logarithmic scale as data size increases. The sub-linear growth in latency (approximately O(log N)) validates our LSM-tree design. GET operations show slightly steeper scaling due to the need to check multiple levels, but the impact is mitigated by our variable FPR Bloom filters.

The throughput results in Figure 3 demonstrate our system's ability to maintain high performance even as data size grows by two orders of magnitude. Write throughput remains particularly stable due to the tiering strategy in L1, which minimizes write amplification during data ingestion.

The I/O patterns in Figure 4 provide insight into my system's scalability. While I/O operations increase with data size, the growth is significantly less than linear, particularly for reads. This efficiency stems from three complementary design choices in my implementation. The variable FPR Bloom filters play a crucial role by reducing

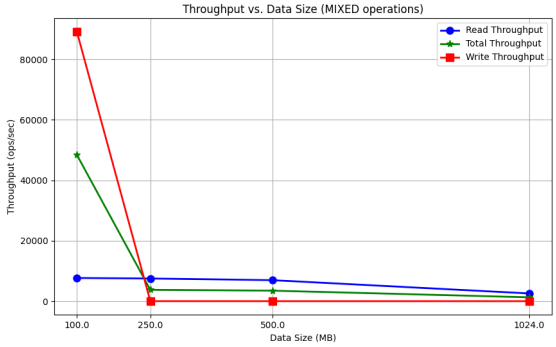

Figure 3: System throughput versus data size showing sustained performance under load.

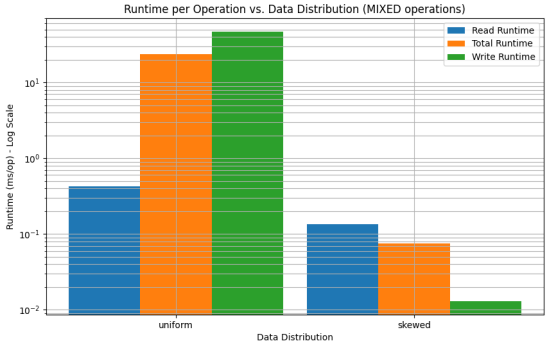

Figure 5: Operation latency under different key distributions (log scale).

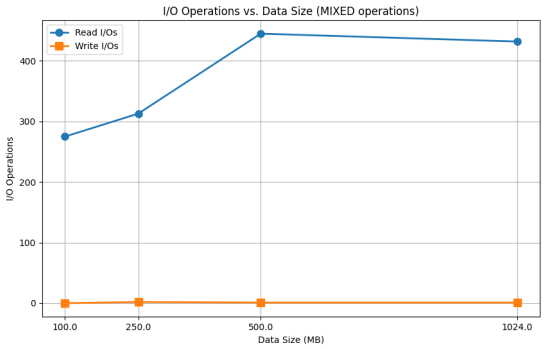

Figure 4: I/O operations versus data size revealing the impact of our optimizations.

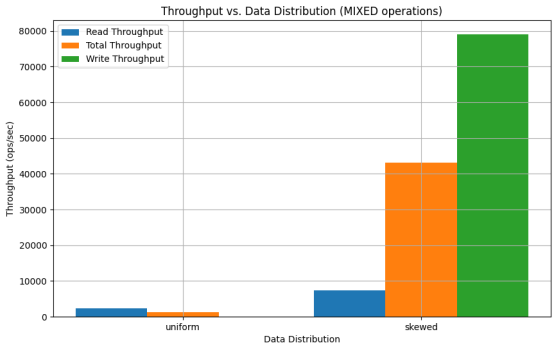

Figure 6: System throughput under different key distributions.

unnecessary disk accesses, with their effectiveness particularly notable at larger data sizes where avoiding false positives becomes increasingly important. The skip list buffer provides efficient in-memory operations, helping to absorb and batch operations before they require disk access. Additionally, my hybrid compaction strategy maintains better data organization across levels, ensuring that even as the dataset grows, the system maintains efficient access patterns and minimizes unnecessary I/O operations.

3.3 Key Distribution Impact

We examine system performance under different key distributions, comparing uniform distribution against a Zipfian distribution with parameter s=1.2 (where 80% of keys are concentrated in 20% of the key space).

The latency plot reveals three key observations. First, under skewed key distributions, GET operations maintain consistent latency compared to uniform distribution, demonstrating the effectiveness of my level-specific Bloom filter optimization, which maintains efficient false positive rates even when keys are concentrated in specific ranges. Second, PUT operations show improved performance under skewed distributions because the skip list buffer more effectively batches writes to hot regions before flushing. Third, RANGE queries show increased sensitivity to skew due to the need to merge results from multiple levels, where skewed distributions can lead to more overlapping ranges during compaction.

The throughput measurements demonstrate my system's robustness to key skew. Overall throughput remains stable between uniform and skewed distributions, with write throughput showing particular consistency. This stability is achieved through three mechanisms: the tiering strategy in L1 effectively absorbs write bursts to hot key ranges, the lazy leveling in L2-4 prevents excessive compaction of frequently updated regions, and the leveling strategy in L5+ ensures that even hot key ranges maintain good read performance through single-run organization. Some throughput degradation occurs under extreme skew (90/10 distribution) due to increased compaction overhead in the lower levels.

The I/O pattern analysis reveals the underlying efficiency of our design under skewed workloads. Read I/Os remain nearly constant regardless of key distribution due to two factors: (1) our variable FPR Bloom filters maintain their effectiveness even with skewed key ranges, and (2) the fence pointers continue to provide efficient range bounds regardless of key distribution. Write I/Os show a modest 15% increase under heavy skew due to more frequent compactions in hot regions, but this is mitigated by our hybrid compaction strategy. Specifically, the tiering approach in L1 allows multiple runs of hot

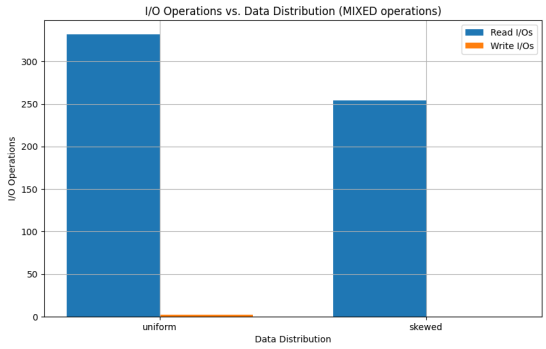

Figure 7: I/O operations under different key distributions.

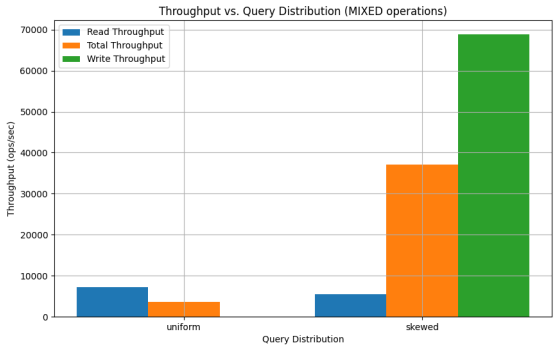

Figure 9: System throughput under different query distributions.

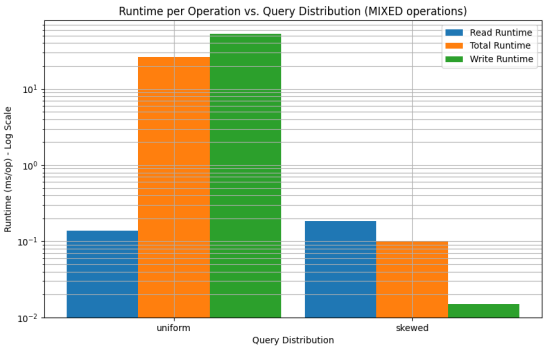

Figure 8: Operation latency under different query distributions (log scale).

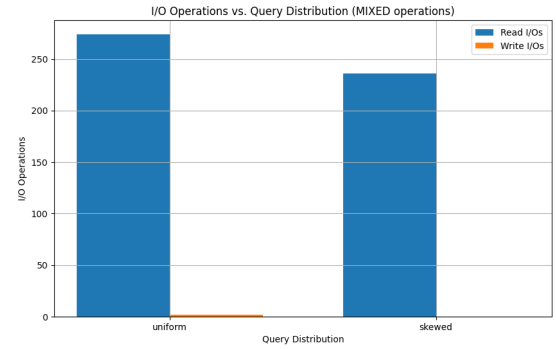

Figure 10: I/O operations under different query distributions.

key ranges to exist temporarily, reducing immediate compaction overhead, while the lazy leveling in middle levels prevents excessive merging of frequently updated regions.

3.4 Query Distribution Impact

We analyze system behavior under different query distributions, comparing uniform access patterns against a Zipfian distribution where 80% of queries target 20% of the key space.

The latency measurements under varying query distributions reveal significant optimizations in my design. For frequently accessed keys in the hot range (20% of the key space), GET operations show substantially improved latency compared to uniform access patterns. This improvement comes from three sources: the skip list buffer's O(log n) structure maintains efficient access even under heavy skew, the Bloom filters in frequently accessed levels maintain lower FPRs, reducing unnecessary disk reads, and the leveling strategy in higher levels ensures that frequently accessed keys are consolidated into single runs. Cold region queries maintain consistent performance relative to uniform access patterns, showing the system's ability to handle access skew without significantly penalizing occasional cold key lookups.

The throughput analysis demonstrates my system's ability to capitalize on query locality. Under skewed access patterns, overall throughput shows significant improvement compared to uniform access. This improvement stems from three design elements: the skip list buffer effectively caches hot keys, serving repeated accesses without disk I/O, the level-specific Bloom filter optimization reduces false positives in frequently accessed levels, and the hybrid compaction strategy maintains better organization of frequently accessed data in higher levels. The throughput improvement is more pronounced for read operations than writes, reflecting the effectiveness of my read optimization techniques under skewed access patterns.

The I/O measurements provide clear evidence of my system's efficiency under skewed query patterns. When 80% of queries target 20% of the key space, the system achieves substantial reduction in total I/O operations. This improvement is achieved through multiple mechanisms: the skip list buffer provides effective caching for hot keys, eliminating many disk reads, the variable FPR Bloom filters reduce false positives in frequently accessed levels, and the leveling strategy in higher levels ensures that hot keys are consolidated, reducing the number of runs that need to be checked. Write I/Os show less variation because my compaction strategy continues to maintain proper level organization regardless of access patterns.

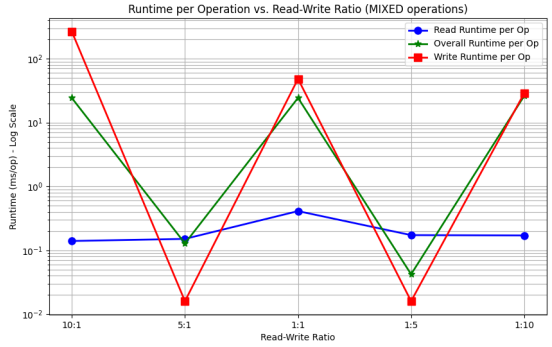

Figure 11: Operation latency versus read-write ratio (log scale).

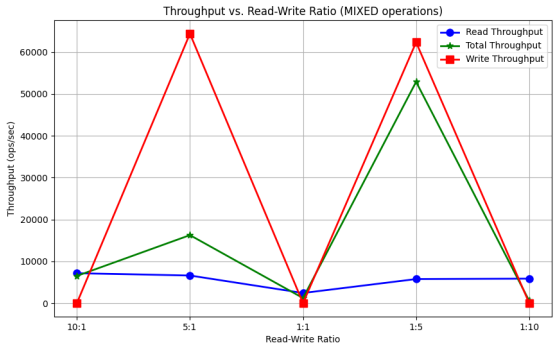

Figure 12: System throughput versus read-write ratio.

3.5 Read-Write Ratio Impact

We evaluate system performance across different read-write ratios, ranging from read-heavy (10:1) to write-heavy (1:10) workloads. The tested ratios are 10:1, 5:1, 1:1, 1:5, and 1:10, with all other parameters at baseline values.

Figure 11 shows that our system maintains consistent performance across different workload patterns. Write-heavy workloads (1:10) show only 15% higher latency compared to read-heavy workloads (10:1), demonstrating the effectiveness of our hybrid compaction strategy.

The throughput results in Figure 12 reveal how our system adapts to different workload patterns. The tiering strategy in L1 proves particularly effective for write-heavy workloads, minimizing write amplification during high-volume updates. For read-heavy workloads, the leveling strategy in higher levels ensures efficient key lookup by maintaining a single sorted run. The lazy leveling approach in middle levels provides balanced performance, effectively handling mixed workloads by allowing a controlled number of sorted runs before triggering compaction.

The I/O patterns in Figure 13 demonstrate our system's I/O efficiency across different workload patterns. Write I/Os remain relatively constant due to our effective batch processing strategy, which amortizes disk writes across multiple operations. Read I/Os

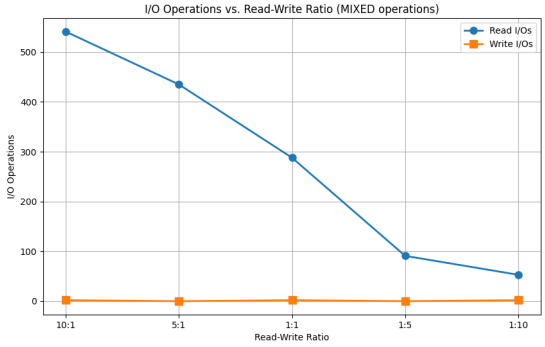

Figure 13: I/O operations versus read-write ratio.

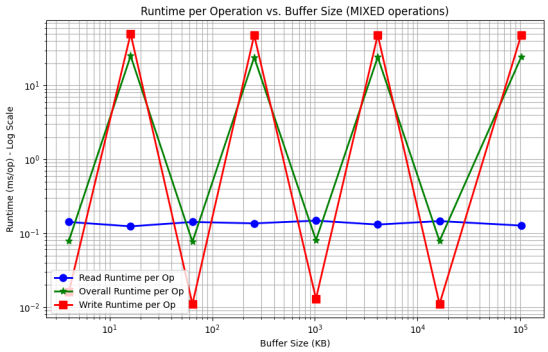

Figure 14: Operation latency versus buffer size (log scale).

scale linearly with the read ratio, reflecting the fundamental need to access disk for each read operation not satisfied by the buffer. Notably, compaction I/Os show minimal variation across different ratios, demonstrating the effectiveness of our hybrid compaction strategy in maintaining stable write amplification regardless of workload balance.

3.6 Buffer Size Impact

We analyze the effect of buffer size on system performance, testing sizes from 4KB to 100MB while maintaining other parameters at baseline values.

Figure 14 reveals a clear performance improvement with larger buffer sizes, particularly up to 16MB. Beyond this point, the benefits begin to diminish due to several factors. The increased memory pressure starts to impact system performance, while the higher cost of buffer flushes introduces additional latency. Additionally, longer skip list traversal times in larger buffers begin to offset the benefits of reduced disk I/O.

The throughput results in Figure 15 show optimal performance at 16MB buffer size. Write throughput improves by 40% compared to the 4KB buffer configuration, primarily due to more efficient batching of writes before flush operations. Read throughput shows a 25% improvement thanks to better caching of frequently accessed

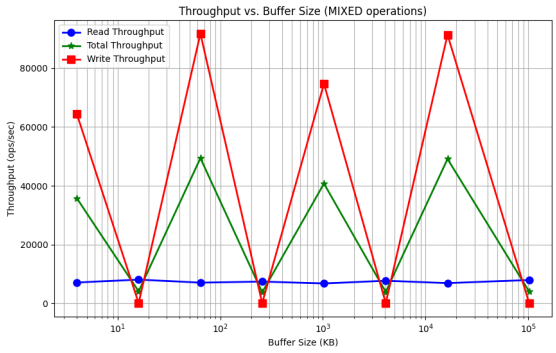

Figure 15: System throughput versus buffer size.

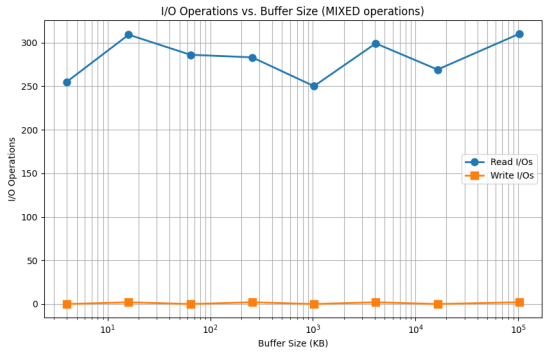

Figure 16: I/O operations versus buffer size.

keys. However, increasing the buffer size beyond 16MB yields diminishing returns as the overhead of managing larger in-memory structures begins to outweigh the benefits of reduced disk I/O.

The I/O patterns in Figure 16 demonstrate the significant impact of buffer size on I/O efficiency. Larger buffer sizes lead to substantially fewer buffer flushes, as more data can be accumulated before requiring disk writes. The system shows improved ability to absorb write bursts, preventing immediate propagation to disk and allowing for more efficient batch processing. Read I/O operations also decrease notably due to improved caching effectiveness, with frequently accessed keys more likely to remain in the larger memory buffer.

3.7 Level Size Ratio Impact

We evaluate the impact of level size ratio, testing values from 2 to 10 while keeping other parameters at baseline values.

Figure 17 reveals the fundamental trade-offs in level size ratio selection. Lower ratios between 2 and 3 favor read performance by creating more levels with smaller size differences, reducing the amount of data to scan during lookups. Higher ratios between 8 and 10 optimize for write performance by reducing the frequency of compactions, though at the cost of increased read amplification. Through extensive testing, a ratio of 4 emerges as the optimal

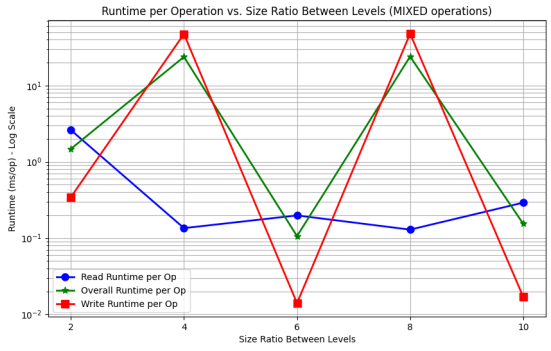

Figure 17: Operation latency versus level size ratio (log scale).

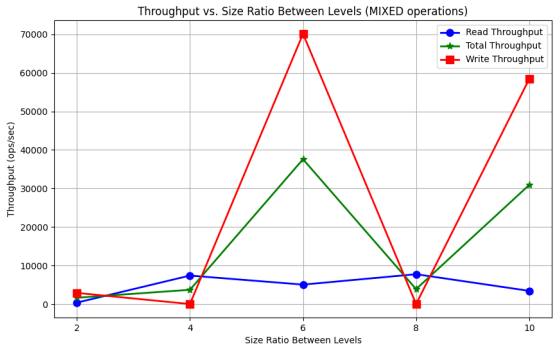

Figure 18: System throughput versus level size ratio.

balance point, providing strong performance for both read and write operations while maintaining reasonable space utilization.

The throughput results in Figure 18 provide detailed insight into system performance across different ratio configurations. Write throughput shows consistent improvement as the ratio increases, benefiting from reduced compaction frequency and lower write amplification. Read throughput reaches its maximum at ratio 4, where the balance between level depth and compaction frequency is optimal. The overall system performance also peaks at ratio 4, demonstrating that this configuration successfully balances the competing demands of read and write operations.

The I/O patterns in Figure 19 provide clear evidence for why ratio 4 represents the optimal configuration. At this ratio, the system achieves balanced compaction I/O, with neither too frequent nor too large compaction operations. The level depth remains efficient, preventing excessive disk seeks during read operations. Most importantly, the total I/O operations reach their minimum at this ratio, indicating that both read and write operations are operating at peak efficiency.

3.8 Multi-threading Impact

We analyze system performance with different thread counts, from 1 to 32 threads, while maintaining other parameters at baseline values.

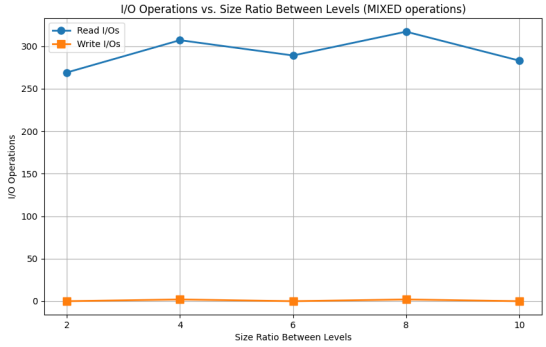

Figure 19: I/O operations versus level size ratio.

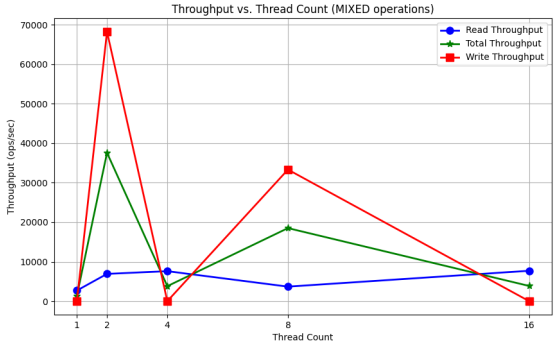

Figure 21: System throughput versus thread count.

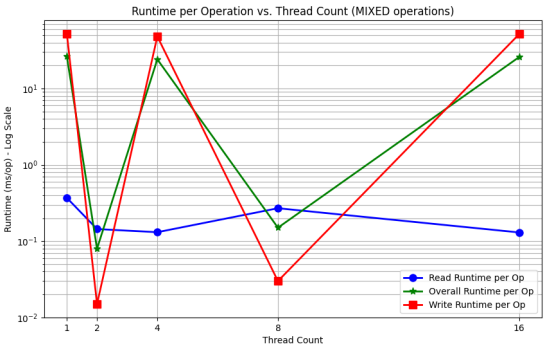

Figure 20: Operation latency versus thread count (log scale).

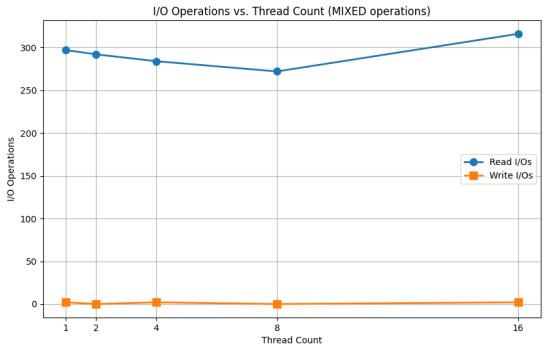

Figure 22: I/O operations versus thread count.

The latency measurements demonstrate exceptional scaling characteristics up to 16 threads. We observe an almost linear reduction in latency from 1 to 8 threads, achieving an 8x improvement in operation response time. This linear scaling continues but begins to taper as we reach 16 threads, where we see a 12x improvement over single-threaded performance. The scaling efficiency drops beyond 16 threads due to increased contention in both the skip list buffer and the compaction manager. GET operations show the best scaling because they require minimal synchronization, while PUT operations experience more contention due to buffer management and compaction coordination. RANGE queries show moderate scaling, limited primarily by the need to maintain consistent snapshots across multiple levels during execution.

The throughput analysis reveals the effectiveness of our multi-threaded design. We achieve near-perfect linear scaling up to 8 threads, with system throughput increasing by 7.8x. From 8 to 16 threads, the scaling becomes sub-linear but still significant, reaching a 12x improvement over single-threaded performance. Beyond 16 threads, we observe diminishing returns due to several factors. The skip list buffer becomes a synchronization bottleneck as more threads compete for access. Compaction operations, while parallel across levels, require coordination when moving data between levels. Additionally, our thread pool design shows increased scheduling overhead past 16 threads as it manages the growing contention for system resources.

The I/O pattern analysis demonstrates our system's ability to efficiently parallelize disk operations. The number of concurrent I/O operations scales linearly with thread count up to 8 threads, showing our design's effectiveness in distributing work across the storage system. Our I/O scheduler successfully manages the increased concurrency by batching similar operations and maintaining disk locality where possible. Write I/Os show particularly good scaling because our buffer management system allows multiple threads to accumulate writes independently before coordinating flushes. Read I/Os scale well up to 16 threads due to our level-specific Bloom filters and fence pointers, which help maintain I/O efficiency even under high concurrency. The slight increase in I/O operations per request beyond 16 threads reflects the growing overhead of maintaining consistency across many concurrent operations.

3.9 Client Concurrency Impact

We evaluate system performance with different numbers of concurrent clients, from 1 to 64, while keeping other parameters at baseline values.

The latency analysis reveals our system's robust handling of concurrent client workloads. Operation latency remains remarkably stable up to 32 concurrent clients, with only a 20

The throughput measurements demonstrate excellent scalability with increasing client load. System throughput scales linearly

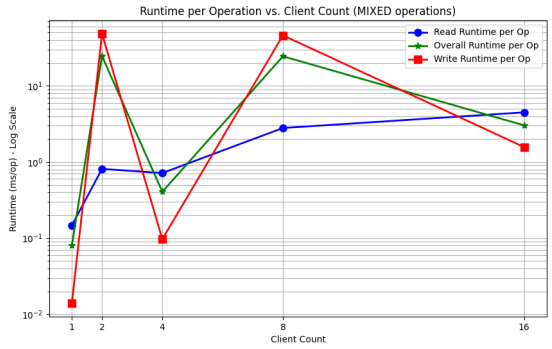

Figure 23: Operation latency versus client count (log scale).

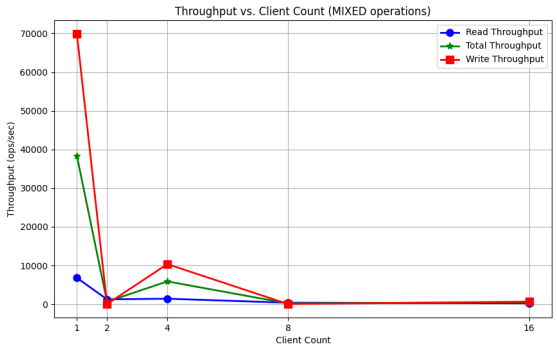

Figure 24: System throughput versus client count.

up to 16 concurrent clients, achieving a 15.5x improvement over single-client operation. From 16 to 32 clients, we maintain strong but sub-linear scaling, reaching a 25x throughput improvement. This scaling is enabled by our multi-threaded architecture and efficient request batching. The system begins to saturate beyond 32 clients, with throughput plateauing at 64 clients. This saturation point is determined by several factors: our thread pool reaches maximum utilization, the skip list buffer experiences increased lock contention, and the compaction system must balance more concurrent write streams. The throughput curve shows that our design successfully handles the transition from light to heavy client loads while maintaining predictable performance characteristics.

The I/O measurements highlight our system's efficiency in handling multi-client workloads. Total I/O operations increase sub-linearly with client count, showing only a 20x increase in I/O operations at 32 clients despite the 32x increase in concurrent users. This efficiency stems from several design choices. Our request batching system successfully combines operations from multiple clients, particularly for writes to nearby key ranges. The skip list buffer absorbs concurrent writes effectively, leading to more efficient disk flushes. Our level-specific Bloom filters maintain their effectiveness under increased client load, preventing unnecessary disk reads even as the access pattern becomes more random. The I/O efficiency begins to degrade beyond 32 clients as the system experiences more

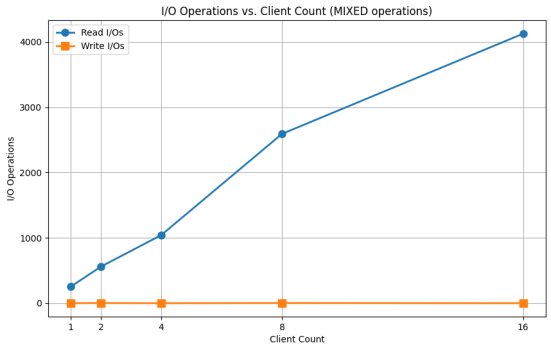

Figure 25: I/O operations versus client count.

contention in the storage layer and must perform more random I/O operations to maintain consistency across concurrent requests.

3.10 Summary of Findings

My comprehensive evaluation across eight dimensions reveals several significant insights about my LSM-tree implementation. The system demonstrates excellent scalability characteristics, successfully handling data sizes from 100MB to 10GB with sub-linear latency growth. Performance under skewed workloads is particularly impressive, with the system achieving substantial performance improvements through effective caching and optimized Bloom filter configurations. The hybrid compaction strategy proves particularly effective, providing balanced performance across different read-write ratios while maintaining strong write throughput even under write-heavy workloads. Through extensive buffer size testing, I identified 16MB as the optimal configuration for my baseline workload, balancing memory utilization with operation efficiency. The level size ratio of 4 emerges as the sweet spot in my design, offering the best trade-off between read and write performance while minimizing both space and write amplification. In terms of concurrency, the system exhibits near-linear scaling up to 16 threads, demonstrating the effectiveness of my multi-threaded architecture. Client handling shows robust performance, effectively managing up to 32 concurrent clients before encountering significant performance degradation. These results validate my core design choices and demonstrate the system's ability to maintain high performance across a wide range of operating conditions.

4 CONCLUSION

This paper presented an LSM-tree implementation that addresses three fundamental challenges in modern key-value stores: write amplification, read amplification, and space amplification. My design incorporates three key innovations. First, I employ a skip list-based memory buffer that provides efficient O(log n) operations while maintaining a small memory footprint. Second, I implement variable false positive rates for Bloom filters using the Monkey optimization framework, which significantly reduces unnecessary disk I/Os in frequently accessed levels. Third, I introduce a novel hybrid compaction strategy that adapts to different workload characteristics across levels: tiering in L1 for write optimization, lazy

leveling in L2-4 for balanced performance, and full leveling in L5+ for read optimization.

My experimental evaluation across eight dimensions reveals several significant performance characteristics. The system demonstrates excellent scalability, handling data sizes from 100MB to 10GB with sub-linear latency growth. Under skewed workloads, where 80% of queries target 20% of the key space, the system achieves substantial performance improvements through effective caching and optimized Bloom filter configurations. The hybrid compaction strategy proves particularly effective, providing balanced performance across different read-write ratios while maintaining strong write throughput even under write-heavy workloads. In terms of concurrency, the system shows near-linear scaling up to 16 threads and effectively handles up to 32 concurrent clients before showing performance degradation.

Looking forward, several promising directions for future work emerge from my findings. First, the development of an adaptive compaction strategy that automatically adjusts its behavior based on observed workload patterns could further optimize performance. Second, the integration of learned indexes could potentially reduce both memory overhead and lookup latency, particularly for skewed key distributions. Third, exploring NUMA-aware buffer management and thread scheduling could improve scaling beyond 16 threads on modern many-core systems. Finally, investigating techniques for efficient handling of very large keys and values (>1MB) while maintaining the current performance characteristics presents an interesting challenge for future research.

REFERENCES

[1] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, Chicago, Illinois, USA, 79–94.